NASA Contractor Report 181779

ICASE REPORT NO. 89-6

ICASE

A LANGUAGE COMPARISON FOR SCIENTIFIC COMPUTING ON MIMD ARCHITECTURES

Mark T. Jones

Merrell L. Patrick

Robert G. Voigt

Contract Nos. NAS1-18107, NAS1-18605
January 1989

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

A Language Comparison for Scientific Computing On MIMD Architectures

Mark T. Jones*, Merrell L. Patrick* and Robert G. Voigt†

Abstract

Cholesky's method for solving banded symmetric, positive definite systems is implemented on a multiprocessor computer using three FORTRAN based parallel programming languages, the Force, PISCES and Concurrent FORTRAN. The capabilities of the languages for expressing parallelism and their user friendliness are discussed, including readability of the code, debugging assistance offered, and expressiveness of the languages. The performance of the different implementations is compared. It is argued that PISCES, using the Force for medium-grained parallelism, is the appropriate choice for programming Cholesky's method on the multiprocessor computer, Flex/32.

*Department of Computer Science, Duke University, Durham, NC 27706

†ICASE, NASA Langley Research Center, Hampton, VA 23665.

This research was supported by the National Aeronautics and Space Administration under NASA contract Nos. NAS1-18107 and NAS1-18605 while the authors were in residence at ICASE. Additional support for the first two authors was provided by NASA grant No. NAG-1-466.

1. Introduction

Efficient programming of parallel computers to support scientific applications is of increasing importance. Although many programming environments are available on different machines, there have been relatively few comparisons of different programming paradigms on the same machine. Several factors that contribute to the useability of a language have been identified. Using these factors this paper explores the strong and weak points of three parallel languages by implementing Choleski's method for solving Ax = b, where A is a banded symmetric positive definite matrix, on the Flexible Computer Corporation Flex/32 [Mat84]. The Flex/32 has twenty processors with each processor having local memory and access to a shared memory. Appendix 9 illustrates the overall architecture of the Flex/32. The architecture and three languages support both shared memory and local memory implementations of the algorithm. In addition, one language supports message passing. Thus, three programming paradigms can be considered: shared memory, message passing, and shared/local which takes advantage of the local memory. These are discussed in the next section. The three languages are all derivatives of FORTRAN and are discussed briefly in section 3. The Choleski algorithm is given in Section 4 along with a brief discussion of the implementation tradeoffs. Section 5 presents observations on the implementation of the algorithm using the various paradigms. The observations are based on factors such as expressibilty of functional parallelism and data partitioning, support for communication and synchronization, runtime cost, ease of program conversion, and user friendliness. The Appendices contain the code representing the implementations.

2. Programming Paradigms

Three different parallel programming paradigms are considered: shared memory, message passing, and share/local (henceforth referred to as local memory). Parallel architectures can also be placed in these three classes. Each paradigm can be implemented on each architecture, but the cost of implementing a paradigm on an architecture that doesn't naturally support that paradigm can be substantial.

For the purposes of this paper, a shared memory architecture is one in which each processor has equal access to a shared or common memory (architectures where processors have cache memory are placed in this category). In a hybrid architecture, each processor has a local memory and access to memory shared by all the processors. Processors in a message passing architecture only have access to local memory and must communicate via messages with other processors.

2.1 Shared Memory

When using the shared memory paradigm, the programmer can view the computer as a sequential computer with several concurrent processes running. Some of the programming issues that arise are similar to those arising in concurrent programming on a sequential machine. Since all processors are viewed as having equal access to all memory, the location of data is not important. However, contention between processors for a particular location in the shared memory or for the interconnection network between the processors and memory must be considered. The programmer is primarily concerned with dividing up the work among the processors to allow for maximum parallelism while minimizing communication and providing synchronization among the processors.

Version A	Version B
	LOCK(sumlock)
sum = sum + 1	sum = sum + 1
	UNLOCK(sumlock)

Figure 1: Shared memory programming bug

All communication and synchronization between processors takes place via shared memory. One of the major burdens that the shared memory paradigm places on the programmer is the necessity to synchronize references to objects that are used by more than one processor. Some objects or sections of code require that they be accessed sequentially and the programmer must ensure that this is the case while trying to keep all the processors doing useful work. This need for synchronization is often the source of "parallel bugs" in shared memory programs ("parallel bugs" are bugs that are introduced because the tasks of the program are being run simultaneously, not traditional programming bugs). This type of bug also arises when running concurrent processes on a sequential computer. Figure 1 shows an example of this type of bug. If several processors are simultaneously executing Version A, more than one processor could fetch the same value for sum, add one to it, and replace sum with the same value. In order to get the correct answer, the addition to sum must be atomic. In version B, the addition to sum is made atomic by putting an exclusive lock around it. This is an example of synchronization which the programmer must provide.

2.2 Message Passing

When programming in the message-passing paradigm, one of the programmer's major concerns is the distribution of data. Since one processor cannot access another processor's memory, performance is improved if the data a processor needs is allocated to its memory. Data exchange and communication between processors is achieved via messages sent explicitly from one processor to another. Thus the programmer is responsible for movement of data and the division of work among processors. The movement of data is achieved by the explicit sending and receiving of messages that contain the data to be moved. Synchronization is implicit in the message passing because a processor does not send data until the data is ready and a processor does not receive data until it is ready to receive it. Thus, the programmer doesn't have to be concerned with the synchronization problem of the shared memory paradigm, but is faced with the new problem of moving data from processor to processor and partitioning this data efficiently across the processors. The programmer must really view this paradigm as a group of isolated processes executing simultaneously that can communicate only by messages, somewhat akin to the communicating sequential processes model of Hoare [Hoa78]. Programs tend to be more difficult to write, but once written, do not have the synchronization bugs that occur in shared memory programs. The code in Figure 1 in the message passing paradigm might look like the code in Figure 2. In this code, each worktask sends the value that is to be added to sum to sumtask which holds sum and is responsible for updating sum. Thus, no explicit synchronization is necessary, just the sending of messages.

```
      sumtask                  worktask
      .                        .
      .                        .
      do 10 i=1,P              send (val) to sumtask
          receive(val)
          sum = sum + val
10    continue
```

Figure 2: Equivalent message passing code for sum problem

2.3 Shared/Local

Programming in the local paradigm is very similar to programming in the shared memory paradigm, with the exception that in order to obtain peak performance, locality of data must be considered. A hybrid architecture can be programmed as a shared memory architecture, but performance may not be optimal because the use of local memory may not be optimal (local references are faster than shared memory references and there is less possibility of contention). The shared/local paradigm lets the programmer make use of this memory hierarchy by allowing the programmer to specify where memory is allocated. After the allocation is done, the program looks the same as a shared memory program. The programmer may also want to make local copies of shared data that a processor accesses many times in order to make fewer shared memory references. The bugs for the shared/local paradigm seem to be the same as for the shared memory paradigm and aside from memory allocation, the code tends to look the same.

3. Languages and Their Use

Languages compared in this study are restricted to FORTRAN based languages that have been implemented on the Flex/32.

3.1 The Force

The Force is a parallel language for shared memory multiprocessors [Jor87]. It consists of extensions to FORTRAN that include constructs for both medium and coarse grained parallelism. A Force is a set of simultaneously initiated processes which run concurrently on different processors. Force members communicate through shared variables and synchronize through barriers and critical regions. Loop iterations are partitioned among Force members by prescheduling or self-scheduling. The Force is currently implemented as a preprocessor to the ConCurrent FORTRAN preprocessor.

3.2 ConCurrent FORTRAN

ConCurrent FORTRAN [Cor86] is a parallel language for the Flex/32 computer implemented by Flexible Computer Corporation. The language assumes a shared memory model of computation with some limited message-passing capabilities for synchronization. The user is responsible for

explicit process management. ConCurrent FORTRAN is implemented as a preprocessor to the FORTRAN compiler.

3.3 PISCES

PISCES is a parallel language and environment for scientific computation [Pra87]. It can support both message-passing based programming and shared memory programming, or a mix of the two. For the purposes of this comparison, the two aspects of PISCES are treated as two separate languages. PISCES is currently implemented as a preprocessor to the FORTRAN compiler and includes a menu-driven environment for configuration of the machine, running the program, and obtaining debugging information. The message-passing portion of PISCES provides facilities for explicit generation of processes and for process identification. It also provides message sending constructs and "handlers" that accept and process messages. The shared memory portion of PISCES is actually the Force language with some minor syntactic differences. All the constructs, including shared variables, of the Force can be used within a PISCES process.

3.4 Using the Languages

Each processor of the Flex 32 Multiprocessor Computer has its own local memory as well as access to a shared memory. This classifies it as a hybrid of distributed and shared memory architectures. Given this hybrid nature and implementations of the three languages which support it, algorithms can have strictly shared memory implementations or local memory implementations which use shared memory for communication amongst processors. In addition, one language, PISCES, supports strictly message passing implementations of the algorithms. Therefore, in our study a total of seven different implementations of Choleski's method were possible on the Flex/32. This makes it a particularly interesting architecture on which to compare the various paradigms for programming parallel computers. In the following sections the terms shared memory, local memory and message passing will be used to distinguish between the different implementations.

4. Choleski's Method and its Parallel Implementation

The solution of

$$Ax = b$$

where A is symmetric positive definite and banded with semi-bandwidth β is carried out in three phases:

1) Factor A into LL^T,
2) forward solve $Ly = b$ for y, and
3) backward solve $L^T x = y$ for x.

There are different ways of organizing each of these phases of computation as described by Dongarra, et al. [DGK84]. For the factorization phase, the "kji" form used by Cleary, et al. [CHO86] has been chosen, namely:

for $k = 1$ to N
  l_{kk}=$a_{kk}^{1/2}$
  for $s = k+1$ to $\min(k+\beta, N)$
    l_{sk}=a_{sk}/l_{kk}
  for $j = k+1$ to $\min(k+\beta, N)$
    for $i = j$ to $\min(k+\beta, N)$
      a_{ij}=a_{ij}-$l_{ik}l_{jk}$

kji Choleski Factorization

This form of Choleski factorization is column oriented, so columns are used to define the granularity of parallelism. Hence, individual processors are assigned sets of columns which they operate upon one at a time. The column wrapped assignment is chosen, which means processor i is assigned columns i, $i+p$, $i+2p$, ..., assuming, of course, there are p processors. In the shared memory versions, each processor operates on its columns which are all stored in shared memory, whereas in the local memory versions a processor's columns are copied to its local memory and operated upon there. In the latter case, data shared by all the processors, e.g., a pivot column, are written to shared memory and accessed there.

For the forward and backward solve phases the inner product (ij) algorithm [RO88] and the column sweep algorithm [GH86] are considered. These are given below.

for $i = 1$ to n
  for $j = \max(i-\beta, 1)$ to $i-1$
    $b_i = b_i$ - $l_{ij}y_j$
  $y_i = b_i$ / l_{ii}

The Inner Product (ij) Algorithm for $Ly = b$

for $j = n$ to 1
  $x_j = y_j$ / l_{jj}
  for $i = j-1$ to $\max(j-\beta, 1)$
    $y_i = y_i$ - $l_{ij}x_j$

Column Sweep (ji) Algorithm for $L^Tx = y$

For the shared memory versions of the forward and backward substitutions, the column sweep algorithm is used in both cases. The inner product algorithm could have been equally as effective. After the factorization phase in the local versions, the columns of L are stored in the local memories in wrapped column form. In this case, the inner product (ij) algorithm for $Ly = b$ and the column sweep (ji) algorithm for $L^Tx = y$ yielded the more efficient implementation. Note that here the hybrid nature of the architecture affected the choice of algorithm used. To optimize use of local memory, the matrix is stored by columns. To take advantage of this storage, the inner product

algorithm followed by the column sweep algorithm must be used, rather than using the column sweep algorithm in the both cases as we did for the shared memory version.

5. Comparisons

In the process of carrying out this study several factors contributing to the useability of a language were identified. These include expressibilty of functional parallelism and data partitioning, support for communication and synchronization, ease of learning the language, ease of converting existing programs, readibilty of the code, debugging and syntax checking, and user friendliness.

As noted above seven different implementations of Choleski's method using the three languages on the Flex/32 are possible. We examine only six of those implementations in carrying out our comparisons below. The six are shared and local memory Force, shared and local memory ConCurrent FORTRAN, strictly message passing PISCES, and PISCES with Force. Programs for each of these implementations are included in the appendices. Note that the PISCES with Force program is just shared memory Force enclosed in a PISCES task definition statement.

5.1 Expression of Functional Parallelism and Data Partitioning

First the expression of functional and data parallelism is examined. In line 1 of the Force program in Appendix 1, a Force macro declares the start of a parallel main program, named Choleski, which will be executed by NP processes each of which will be identified by a unique identifier ME. The number of processes executing the program is a parameter specified by the user at runtime. A "driver" routine creates these processes, assigns values to NP and ME and returns control to the user main program. All processes begin executing from this point on, until they are terminated by the Join statement in line 141. Segments of program which are to be executed by only one process are enclosed in a Barrier - End Barrier pair, e.g., the program segment which puts the pivot column into shared memory for everyone to access (lines 70 - 74). Without barriers each process would execute the main program (the function, in this case) in parallel.

Another example of functional parallelism is illustrated by the parallel Presched DO loop in lines 38-40 of the shared memory version of the Force in Appendix 2. Since the statements within the loop indexed by S do not depend on each other, they can be executed in parallel for different values of S. Pre-scheduling partitions different values of S evenly over processes at compile time. The function being executed in parallel is the computation of the pivot column.

In ConCurrent FORTRAN, the Process statement defines a process to the executing environment and if the statement is within a COBEGIN or COBLOCK statement, it also starts execution of the process. For example, in lines 71-75 of Appendix 3, NP processes are defined where NP is the number of processors being used. Since the process statements are in a COBLOCK statement, each process will begin execution of the Choleski factorization subroutine ELCOL() at the end of the COBLOCK statement. Process with tag PID(I) will be executed by processor number PROCNUM(I) and will operate upon the set of columns assigned to it's local memory by the processes executed in the COBLOCK statements 62-66. This set of statements accomplishes the data partitioning needed for parallel execution of the Choleski factorization given in lines 152-187 (the main body of the subroutine ELCOL).

Every PISCES program is structured as a set of one or more tasks that carry out the computational work. The first statement in the PISCES program of Appendix 5 defines the main task,

chol. Within this parent task other tasks are initiated which will work in parallel to carry out the Choleski factorization, the forward solve and backward solve. These tasks are initiated in statement 193 with statements defining the Choleski factorization phase of the tasks given in lines 263-301. Sets of data required by the tasks are sent to them at task initiation time much as data is passed to a FORTRAN subroutine when it is called. Subtask initiation and the passing of data to them are illustrated in lines 82-92 of Appendix 5.

The Force constructs provide the user with the ability to do medium grain, loop-level parallelism (using the parallel do loops) as well as coarser grain parallelism by simply calling subroutines within the parallel do loops. These levels of parallelism are supported efficiently by starting up processes on each processor at the beginning of the program and using constructs like the Barrier statement to provide synchronization. With PISCES and ConCurrent FORTRAN, the user is responsible for starting up the processes and is limited to a coarser grain granularity unless he provides the synchronization constructs. The implementation of Choleski factorization required loop-level parallelism. This required a high ratio of messages to computation in the case of PISCES and the use of the WHEN and CFlock statements in ConCurrent FORTRAN to construct the equivalent of a barrier.

5.2 Communication

Here language features and constructs which support the communication of intermediate data between tasks or processes executing in parallel are compared.

Within the Force program of Appendix 1 and the ConCurrent FORTRAN program of Appendix 3, communication between processes is accomplished by a process assigning the values to be communicated into shared variables in shared memory from which they can be read by other processes which need them. This is illustrated, e.g., within the Choleski factorization loop, given by lines 55-85 in Appendix 1 and lines 152-185 in Appendix 3, where the process owning the current pivot column will modify it and then write it from it's private local memory to a shared variable in shared memory. This action is carried out by a simple assignment statement. The Force shared memory program required no communication between the tasks.

In PISCES programs, the communication of intermediate data between executing tasks is more explicit. This is accomplished with "send" statements and "accept" statements which use "handlers" to accept the data being sent. The use of these constructs is illustrated in the Choleski factorization tasks, lines 251-289 of Appendix 5. If a task owns the current pivot column it updates it and uses the "to all send" statement to send it to all other tasks. The send statement also specifies the name of a "handler" pivot in this case, which accepts the data. Statements 268-276 deal with the acceptance of the pivot column while statements 373-385 define the "handler" task.

The setup time for communication (and programming time) required by PISCES is much larger than that of the local memory versions of Force and ConCurrent FORTRAN. In Force and ConCurrent FORTRAN, it is a simple matter of using an assignment statement to assign data to a variable in shared memory and then the other processors can read this data. In PISCES, the programmer must use a send statement to send the message to the tasks that need the data, and those tasks must then execute a "handler" which is in effect a subroutine.

5.3 Synchronization

Next, the constructs available in the different languages for managing synchronization of processes and tasks are examined. Two types of synchronization are used within the Force program of Appendix 1, the barrier and critical statements. The use of the barrier statement is illustrated in the Choleski factorization loop. Statements 70 and 74 are a "Barrier" - "end Barrier" pair. This causes all processes to wait before proceeding until the process which computes the current pivot column has written it to shared memory. The use of the critical section is illustrated in lines 100-102 of Appendix 1.

In the ConCurrent FORTRAN program of Appendix 3, the WHEN statement and CFlock statements are used to accomplish synchronization. The WHEN statement appears in line 162 and prevents the process which owns the current pivot column from updating it and writing it to shared memory until all other processes have finished using the old pivot column. The WHEN statement in line 170 prevents the processes that need the current pivot column from continuing until it is available in shared memory. The CFlock-CFulck statement in lines 182-184 assures that only one process at a time will update the shared memory variable, NUMDONE.

In the PISCES program of Appendix 5, "send" and "accept" statements are used to synchronize the execution of tasks. For example, in the Choleski factorization, a task cannot update it's set of columns until it has accepted the pivot column (lines 268-270) from the task which owns, updates and sends it (lines 254-260). A check is made by each task to see that the pivots it requires are being received in proper order. If not, the task resends them to itself until they are received in the proper order (lines 273-275).

When using PISCES message passing, synchronization is taken care of by the communication of data; the programmer is not responsible for it. However, in the ConCurrent FORTRAN and Force programs this is one of the programmer's main responsibilities. The Force synchronization constructs are easier to use than those in ConCurrent FORTRAN, but they are not as flexible. The Barrier statement is very useful, however it requires that *all* processors reach a Barrier. The programmer cannot specify that one task execute some code while the other tasks execute some other code that contains a Barrier. When the programmer needs the equivalent of a barrier statement in ConCurrent FORTRAN he must construct it himself.

5.4 Runtime Cost

Comparisons of the runtimes of the various programs were obtained by running the programs on several different data sets. Appendix 7 shows the results of this comparison on a data set generated from a structural analysis application at NASA Langley Research Center. Negative speedups occur in some of the forward and back solve cases due to the large ratio of synchronization to computation in these algorithms. From these comparisons, it is clear that ConCurrent FORTRAN becomes increasingly costly as more processors are added. The Force versions are faster, with the shared and local memory versions being competitive with each other. The difference in execution times of the Force programs and strictly message passing PISCES programs is due in part to the overhead inherent in message passing and in part to its implementation on an architecture which does not support message passing. Runtimes of Force and PISCES with Force programs are nearly identical. The high cost of ConCurrent FORTRAN is due to the costly implementation of WHEN on the Flex/32 compared to the efficient lock routines used in Force.

5.5 Conversion of Existing Programs

If the parallelism in an existing FORTRAN program exists in DO-loops then it is a fairly simple matter to convert FORTRAN into the Force by using pre-scheduled or self scheduled loops. Synchronization is accomplished by barrier statements and critical sections which are easy to use. In both PISCES and ConCurrent FORTRAN, a conversion of existing programs involves more restructuring of the code with PISCES requiring considerably more than ConCurrent FORTRAN. One measure of coding efficiency is the number of lines of code. By this measure, as seen in Appendix 8, the Force is clearly the language of choice of the three languages examined for conversion of existing FORTRAN code.

5.6 Readability and Learning of the Languages

By design, the Force is like FORTRAN with a small number of constructs added. The use of these constructs is reasonably intuitive. Hence, programmers who know FORTRAN can easily learn and read the Force. This can be observed by looking at the Force program of Appendix 2. Although FORTRAN based, PISCES is harder to learn. First, the language is based on the idea of communicating tasks which is a programming paradigm quite different from that of standard languages. Because of this, the new constructs are more complex and hence more difficult to learn. They are, however, much more versatile than those in the Force and ConCurrent FORTRAN. A comparison of the Force program in Appendix 1 with the PISCES program of Appendix 5 clearly indicates different complexities of the two languages. The constructs added to FORTRAN to produce ConCurrent FORTRAN are not much more complex than those those added to the Force.

The readability of a program written in some language is, of course, related to the ease with which that language can be learned. It is not surprising then, given knowledge of FORTRAN, that a Force program is relatively easy to read. Force constructs are simple and almost self-explanatory. However, the lack of explicit process management can create difficulty in understanding the flow of program control in a Force program. For example, in the factorization portion of the Force program in Appendix 1 (lines 55-85), every processor is executing the same code and it is difficult to follow the flow of control.

Once one understands how processes are initiated and the meaning of "when" and "lock/unlock" statements, ConCurrent FORTRAN is quite readable. As PISCES is more difficult to learn, PISCES programs are more difficult to read. PISCES parallel constructs are quite complex, e.g., the message handlers of PISCES tend to hide some of the work being done in a task. This is illustrated by examining statements 257-259 of the PISCES program of Appendix 5 where the "accept" statement names a "handler" incol. One must locate the code for the "handler" incol, lines 370-382, which is not very self-explanatory.

A reasonable measure of difficulty of reading (and time taken to write) languages is comparing the number of lines of code for the same implementation in different languages. This would not always be a good measure of readability if we were comparing very different languages such as APL and FORTRAN, however, since the languages being discussed are all extensions to FORTRAN, it appears to be reasonable. Appendix 8 shows the comparison based on the lines of code. It is clear the Force is the least verbose of the languages and that local versions take more lines of code than shared versions. This is illustrated by comparing the Force local memory and shared memory versions of the programs in Appendix 1 and Appendix 2, respectively. First, one observes that the number and type of declaration statements increases. In the local memory version, additional lines

of code (44-53) are needed to distribute data to the local memories. Also extra code is needed in each of the factoring, forward solve and backward solve phases of solution, e.g., in the factoring phase of the local memory version a test is made (statement 60) to see which processor owns the pivot column; it then computes it and places it in shared memory.

5.7 Debugging and Syntax Checking

All three languages suffer from the problem that they are preprocessors, so the FORTRAN syntax errors that are detected by the FORTRAN compiler have line numbers that do not match the line numbers of the original source file. The programmer must therefore look at the output of the preprocessor to find his syntax errors. The Force preprocessor gives no information on syntax errors that involve Force constructs, it simply passes them on to the compiler. It also provides no runtime debugging support. PISCES will detect many of the syntax errors involving PISCES constructs and give the correct line numbers of the errors in the source file. PISCES also provides very good runtime debugging support, with the capability to trace all messages, process starts, etc. ConCurrent FORTRAN will detect many syntax errors involving ConCurrent constructs and will give the correct line numbers of the errors in the source file. However, it provides no runtime debugging support.

5.8 User Friendliness

To help the user, the Force provides a routine called Forcerun that will allow the user to specify the name of a program to run and the number of processes to be used in running it. This program therefore masks any of the hardware details from the user and is the same for every machine on which the Force is implemented. PISCES is more "user friendly"; it allows the user to interactively configure the machine, set trace options, and run the program. During the run it interactively allows the user to examine such things as message queues and memory being used. ConCurrent FORTRAN, on the other hand has none of the user friendly features of the other two.

6. Conclusions

The above discussion focused on comparing the Force, ConCurrent FORTRAN and PISCES as parallel programming languages. As indicated in the Appendices, the local and shared memory versions of the Force programs are very similar; there is a small difference in the performance of the two codes due to architectural characteristics of the Flex/32. It should be added that PISCES has incorporated all the features of the Force within it's environment. Hence one is able to use the best features of both PISCES and the Force when writing programs using PISCES. Of course, resulting programs can look like nearly pure PISCES programs, nearly pure Force programs or anywhere between. The PISCES Force program is nearly the same as the Force program but is enclosed in a PISCES task which provides the richness of the PISCES environment for debugging and testing the program. Performance results given in Appendix 7 indicate that PISCES Force performs equally as well as the Force program. We therefore conclude that the best implementation of Choleski's method on the Flex/32 is one which uses PISCES with Force constructs.

Clearly much progress in needed in the area of parallel languages for scientific computing. One approach is to construct a FORTRAN-based language that allows the easy expression of the parallelism inherent in an algorithm and provides a reasonable amount of portability across

architectures. A difficulty in this area is that many of the parallel architectures are very different from each other. There is a question of just how much portability can be achieved without an unreasonable loss in efficiency.

7. Acknowledgements

The authors thank Harry Jordan and Terry Pratt for many helpful conversations during the writing of this paper.

References

[CHO86] Andrew J. Cleary, David L. Harrar II, and James M. Ortega. Gaussian elimination and Choleski factorization on the FLEX/32. Technical Report Applied Mathematics Report No. RM-86-13, University of Virginia, December 1986.

[Cor86] Flexible Computer Corporation. *ConCurrent FORTRAN(TM) Reference Manual*, 1986.

[DGK84] J.J. Dongarra, F.G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, January 1984.

[GH86] A. Geist and M. Heath. Matrix factorization on a hypercube multiprocessor. In M. Heath, editor, *Hypercube Multiprocessors*, pages 161–180. SIAM, Philadelphia, 1986.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Jor87] Harry F. Jordan. The force. In Jameson, Gannon, and Douglas, editors, *Characteristics of Parallel Algorithms*, chapter 16. MIT Press, 1987.

[Mat84] N. Matelin. The FLEX/32 multicomputing environment. In R.J. Hayduk and A.K. Noor, editors, *Research on Structures and Dynamics*. NASA, 1984. NASA CP-2335.

[Pra87] Terrence W. Pratt. PISCES 2 user's manual. Icase interim report 2, ICASE, NASA Langley Research Center, 1987.

[RO88] C. Romine and J. Ortega. Parallel solution of triangular systems of equations. *Parallel Computing*, 6:109–114, 1988.

Appendix 1: Force - local memory version

```
1         Force Choleski of NP ident ME
2         Shared INTEGER Beta,BetaP,N
3   C      Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
4         Private INTEGER  I,J,K,S
5         Private INTEGER tempmin
6         Private INTEGER tempmax
7         Shared REAL TempA(100000)
8   C      TempA is temporary holding for the matrix
9         Private REAL A(100000)
10  C      A contains the matrix
11        Private INTEGER Assign(2000)
12  C      Assign is the array of column numbers this processor owns
13        Private INTEGER NumCols
14  C      NumCols is the number of columns owned
15        Shared REAL CurL(2000)
16  C      The current pivot column
17        Shared REAL tempCurL(2000)
18  C      The temporary var holding the next pivot column
19        Shared REAL RHS(2000)
20  C      RHS is the right-hand side vector
21        Private REAL PriSUM
22        Shared LOGICAL UPDTRH
23  C      UPDTRH is used for a critical section
24        Shared REAL Y(2000)
25  C      The Y vector in the forward solve
26        Shared REAL X(2000)
27  C      X is the solution vector
28        Private INTEGER LCol, L2Col
29        End declarations
30
31        Barrier
32  C        Decide whether to read in or build the matrix
33          READ(9,525) I
34          IF (I.eq.0) THEN
35            CALL INITMAT(TempA,RHS,N,Beta,BetaP)
36          ELSE
37            CALL CRMAT(TempA,RHS,N,Beta,BetaP)
38          END IF
39          WRITE(6,600) N, Beta
40  600      FORMAT(' Order',I4,' matrix with a semi-bandwidth',I4,'.')
41  525      FORMAT(I4)
42        End Barrier
43  C      Transfer the matrix from Shared memory to local memory
44        LCol = 1
45        Presched DO 700 I = 1, N
46          DO 710 J = 1, BetaP
47            A((LCol*BetaP)+J) = TempA((I*BetaP)+J)
48  710      Continue
49          Assign(LCol) = I
50          LCol = LCol + 1
51  700    End Presched DO
52        NumCols = LCol - 1
53
```

```
54 C     Start the choleski factorization loop
55       LCol = 1
56       DO 100 K = 1, N
57         tempmin = min(K+Beta,N)
58 C       if this processor owns the pivot column then compute it and
59 C       place it in shared memory
60         IF (Assign(LCol).eq.K) THEN
61           A((LCol*BetaP)+1) = sqrt(A((LCol*BetaP)+1))
62           tempCurL(1) = A((LCol*BetaP)+1)
63           DO 110 S = K + 1, tempmin
64             A((LCol*BetaP)+S-K+1) = A((LCol*BetaP)+S-K+1) /
65     C             A((LCol*BetaP)+1)
66             tempCurL(S-K+1) = A((LCol*BetaP)+S-K+1)
67 110       Continue
68           LCol = LCol + 1
69         END IF
70         Barrier
71           DO 115 S = K, tempmin
72             CurL(S-K+1) = tempCurL(S-K+1)
73 115       Continue
74         End Barrier
75 C       Update the rest of the columns
76         DO 120  L2Col = 1, NumCols
77           J = Assign(L2Col)
78           IF ((J.ge.K+1).and.(J.le.tempmin)) THEN
79             Do 130 I = J, tempmin
80               A((L2Col*BetaP)+I-J+1) = A((L2Col*BetaP)+I-J+1)
81     C               - CurL(I-K+1)*CurL(J-K+1)
82 130         CONTINUE
83           END IF
84 120     CONTINUE
85 100   CONTINUE
86
87 C     Forward Solve (using inner product)
88       LCol = 1
89       DO 300 I = 1, N
90         tempmax = max(I-Beta,1)
91         PriSUM = 0
92 C       Compute the amount this processor will subtract from the RHS
93         DO 310 L2Col = 1, NumCols
94           J = Assign(L2Col)
95           IF ((J.ge.tempmax).and.(J.le.I-1)) THEN
96             PriSUM = PriSUM + A((BetaP*L2Col)+I-J+1)*Y(J)
97           END IF
98 310     CONTINUE
99 C       Update the RHS
100        Critical UPDTRH
101          RHS(I) = RHS(I) - PriSUM
102        End Critical
103        IF (I.eq.Assign(LCol)) THEN
104          CurDiv = A((BetaP*LCol)+1)
105          LCol = LCol + 1
106        END IF
107        Barrier
108          Y(I) = RHS(I) / CurDiv
```

```
109          End Barrier
110 300     CONTINUE
111
112 C       Backward Solve (using col-sweep)
113         LCol = NumCols
114         DO 400 J = N, 1, -1
115 C          If we own column J, then compute the new X
116           IF (J.eq.Assign(LCol)) THEN
117             X(J) = Y(J) / A((BetaP*LCol)+1)
118             LCol = LCol - 1
119             IF (LCol.eq.0) LCol = 1
120           END IF
121           Barrier
122           End Barrier
123           tempmax = max(J-Beta,1)
124 C         Everyone update Y
125           DO 410 L2Col = NumCols, 1, -1
126             I = Assign(L2Col)
127             IF ((I.le.J-1).and.(I.ge.tempmax)) THEN
128               Y(I) = Y(I) - A((BetaP*L2Col)+J-I+1)*X(J)
129             END IF
130 410       CONTINUE
131 400     CONTINUE
132
133 C       Print the solution vector
134         Barrier
135         DO 500 J = 1, N
136           WRITE(8,680) J, X(J)
137 500     CONTINUE
138 680     FORMAT(' X(',I4,') = ',6E13.6)
139         End Barrier
140
141         Join
142         END
```

Appendix 2: Force - shared memory version

```
1           Force Choleski of NP ident ME
2           Shared INTEGER Beta,BetaP,N
3   C       Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
4           Private INTEGER  I,J,K,S
5           Private INTEGER tempmin
6           Private INTEGER tempmax
7           Shared REAL A(100000)
8   C       A contains the matrix
9           Shared REAL RHS(2000)
10  C       RHS is the right-hand side vector
11          Shared REAL Y(2000)
12  C       The Y vector in the forward solve
13          Shared REAL X(2000)
14  C       X is the solution vector
15          End declarations
16
17          Barrier
```

```
18 C       Decide whether to read in or build the matrix
19         READ(9,525) I
20         IF (I.eq.0) THEN
21           CALL INITMAT(A,RHS,N,Beta,BetaP)
22         ELSE
23           CALL CRMAT(A,RHS,N,Beta,BetaP)
24         END IF
25         WRITE(6,600) N, Beta
26 600     FORMAT(' Order',I4,' matrix with a semi-bandwidth',I4,'.')
27 525     FORMAT(I4)
28       End Barrier
29
30 C     Start the choleski factorization loop
31       DO 100 K = 1, N
32 C       Compute the first element of the pivot column
33         Barrier
34           A((K*BetaP)+1) = sqrt(A((K*BetaP)+1))
35         End Barrier
36         tempmin = min(K+Beta,N)
37 C       Compute the rest of the pivot column
38         Presched DO 110 S = K+1, tempmin
39           A((K*BetaP)+S-K+1) = A((K*BetaP)+S-K+1) / A((K*BetaP)+1)
40 110     End Presched DO
41         Barrier
42         End Barrier
43 C       Update the rest of the columns
44         Presched DO 120  J = K+1, tempmin
45           Do 130 I = J, tempmin
46             A((J*BetaP)+I-J+1) = A((J*BetaP)+I-J+1)
47     C                          - A((K*BetaP)+I-K+1)*A((K*BetaP)+J-K+1)
48 130       CONTINUE
49 120     End Presched DO
50 100   CONTINUE
51
52 C     The foward solve (using col-sweep)
53       DO 300 J = 1, N
54         Barrier
55         Y(J) = RHS(J) / A((BetaP*J)+1)
56         End Barrier
57         tempmin = min(J+Beta,N)
58         Presched DO 310 I = J+1, tempmin
59           RHS(I) = RHS(I) - A((BetaP*J)+I-J+1)*Y(J)
60 310     End Presched DO
61 300   CONTINUE
62
63
64 C     The backward solve (using col-sweep)
65       DO 400 J = N, 1, -1
66         Barrier
67         X(J) = Y(J) / A((BetaP*J)+1)
68         End Barrier
69         tempmax = max(J-Beta,1)
70         Presched DO 410 I = J-1, tempmax, -1
71           Y(I) = Y(I) - A((BetaP*I)+J-I+1)*X(J)
72 410     End Presched DO
```

```
73  400   CONTINUE
74
75  C     Print out the solution vector
76        Barrier
77        DO 500 J = 1, N
78          WRITE(8,680) J, X(J)
79  500   CONTINUE
80  680   FORMAT(' X(',I4,') = ',6E13.6)
81        End Barrier
82
83        Join
84        END
```

Appendix 3: ConCurrent FORTRAN - local memory version

```
1         PROGRAM MAIN
2         Shared INTEGER /label1/ PRCNUM(20)
3   C     PRCNUM holds the physical proc number corresponding the
4   C     the logical proc number
5         Shared INTEGER /label2/ NP
6   C     NP is the number of processors
7         Shared INTEGER /label3/ NUMDONE
8         Shared INTEGER /label4/ Beta,BetaP,N
9   C     Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
10        Shared INTEGER /label5/ PIVCOL
11        Shared REAL /label6/ TempA(30000)
12        REAL A(30000)
13  C     A contains the matrix
14        common /pblk1/ A(30000)
15        INTEGER ASSIGN(500)
16  C     Assign contains the list of columns that each processor owns
17        common /pblk2/ ASSIGN(500)
18        INTEGER NUMCOLS
19  C     numcols is the number of columns that a processor owns
20        common /pblk3/ NUMCOLS
21        Shared REAL /label12/ CurL(2000)
22  C     the current pivot column
23        Shared REAL /label7/ RHS(2000)
24  C     RHS is the right-hand side vecto
25        Shared REAL /label8/ Y(2000)
26  C     The Y vector in the forward solve
27        Shared REAL /label9/ X(2000)
28  C     X is the solution vector
29        Shared CHARACTER /label11/ NUMLCK
30        EXTERNAL LOADC
31        EXTERNAL ELCOL
32        EXTERNAL FORW
33        EXTERNAL BACK
34        INTEGER PID(20)
35        INTEGER  I
36        INTEGER  ICFret
37        INTEGER  tempmax, tempmin
38
39  C     Allocate a lock
```

```
40        CALL CFgetl(ICFret,'NUMLCK')
41        open(unit=2,cpu=1,file='/usr/u1/mtj/concur/choleski/param.dat')
42  C      Read in the number of processors
43        READ(2,525) NP
44        PRINT *, ' Using ',NP,' processors'
45        DO 15 I = 1, NP
46          PRCNUM(I) = I + 2
47  15    CONTINUE
48
49  C      Decide whether to read in or build the matrix
50        READ(2,525) I
51        IF (I.eq.0) THEN
52          CALL INITMAT(TempA,RHS,N,Beta,BetaP)
53        ELSE
54          CALL CRMAT(TempA,RHS,N,Beta,BetaP)
55        END IF
56        WRITE(6,600) N, Beta
57  600   FORMAT(' Order',I4,' matrix with a semi-bandwidth',I4,'.')
58  525   FORMAT(I4)
59
60  C      Load up the private copies of TempA
61        PRINT *, ' Making private copies'
62        COBLOCK
63          DO 155 I = 1, NP
64            PROCESS(PID(i),LOADC(),PRCNUM(I))
65  155       CONTINUE
66        END COBLOCK
67
68        PIVCOL = 0
69  C      Start the factorization processes on each processor
70        NUMDONE = NP
71        COBLOCK
72          DO 150 I = 1, NP
73            PROCESS(PID(i),ELCOL(),PRCNUM(I))
74  150       CONTINUE
75        END COBLOCK
76
77        PIVCOL = 0
78  C      Start the forward solve processes on each processor
79        NUMDONE = 0
80        COBLOCK
81          DO 160 I = 1, NP
82            PROCESS(PID(i),FORW(),PRCNUM(I))
83  160       CONTINUE
84        END COBLOCK
85
86        PIVCOL = N + 1
87  C      Start the back solve processes on each processor
88        NUMDONE = NP
89        COBLOCK
90          DO 170 I = 1, NP
91            PROCESS(PID(i),BACK(),PRCNUM(I))
92  170       CONTINUE
93        END COBLOCK
94
```

```
95  C     print the solution vector
96          DO 500 J = 1, N
97            WRITE(8,680) J, X(J)
98  500     CONTINUE
99  680     FORMAT(' X(',I4,') = ',6E13.6)
100
101         CALL CFkill(ICFret,0)
102         END
103
104 C     private copies task
105         SUBROUTINE LOADC()
106         Shared REAL /label6/ TempA(30000)
107         REAL A(30000)
108         common /pblk1/ A(30000)
109         INTEGER ASSIGN(500)
110         common /pblk2/ ASSIGN(500)
111         INTEGER NUMCOLS
112         common /pblk3/ NUMCOLS
113         INTEGER plself
114         INTEGER MYNUM
115         INTEGER I, J
116         Shared INTEGER /label2/ NP
117         Shared INTEGER /label4/ Beta,BetaP,N
118
119         MYNUM = plself()
120         NUMCOLS = 0
121         DO 10 I = MYNUM, N, NP
122           NUMCOLS = NUMCOLS + 1
123           ASSIGN(NUMCOLS) = I
124           DO 20 J = 1, BetaP
125             A((NUMCOLS*BetaP)+J) = TempA((I*BetaP)+J)
126 20        CONTINUE
127 10      CONTINUE
128
129         RETURN
130         END
131
132 C     factorization task
133         SUBROUTINE ELCOL()
134         INTEGER K,I,J,S
135         Shared INTEGER /label2/ NP
136         Shared INTEGER /label3/ NUMDONE
137         Shared INTEGER /label4/ Beta,BetaP,N
138         Shared INTEGER /label5/ PIVCOL
139         Shared CHARACTER /label11/ NUMLCK
140         INTEGER ICFret
141         INTEGER MYPIV, MYPIV2
142         INTEGER tempmin
143         REAL A(30000)
144         common /pblk1/ A(30000)
145         INTEGER ASSIGN(500)
146         common /pblk2/ ASSIGN(500)
147         INTEGER NUMCOLS
148         common /pblk3/ NUMCOLS
149         Shared REAL /label12/ CurL(2000)
```

```
150
151 C     Start the choleski factorization loop
152         MYPIV = 1
153         DO 100 K = 1, N
154           tempmin = min(K+Beta,N)
155 C          If I own column K then compute the pivot col
156           IF (K.eq.ASSIGN(MYPIV)) THEN
157             A((MYPIV*BetaP)+1) = sqrt(A((MYPIV*BetaP)+1))
158             DO 110 S = K+1, tempmin
159               A((MYPIV*BetaP)+S-K+1) = A((MYPIV*BetaP)+S-K+1)
160 &                 / A((MYPIV*BetaP)+1)
161 110          CONTINUE
162             WHEN (NUMDONE.eq.NP) CONTINUE
163             DO 115 S = K, tempmin
164               CurL(S-K+1) = A((MYPIV*BetaP)+S-K+1)
165 115          CONTINUE
166             MYPIV = MYPIV + 1
167             NUMDONE = 0
168             PIVCOL = PIVCOL + 1
169           ELSE
170             WHEN (PIVCOL.eq.K) CONTINUE
171           ENDIF
172 C         Update the rest of the columns
173           DO 120  MYPIV2 = 1, NUMCOLS
174             J = ASSIGN(MYPIV2)
175             IF ((J.ge.K+1).and.(J.le.tempmin)) THEN
176               Do 130 I = J, tempmin
177                 A((MYPIV2*BetaP)+I-J+1) = A((MYPIV2*BetaP)+I-J+1)
178     C               - CurL(I-K+1)*CurL(J-K+1)
179 130           CONTINUE
180             ENDIF
181 120        CONTINUE
182           CALL CFlock(ICFret,1,'NUMLCK')
183           NUMDONE = NUMDONE + 1
184           CALL CFulck(ICFret,1,'NUMLCK')
185 100     CONTINUE
186         RETURN
187         END
188
189 C     the foward solve task using inner-product
190         SUBROUTINE FORW()
191         INTEGER I,J
192         REAL A(30000)
193         common /pblk1/ A(30000)
194         INTEGER ASSIGN(500)
195         common /pblk2/ ASSIGN(500)
196         INTEGER NUMCOLS
197         common /pblk3/ NUMCOLS
198         Shared INTEGER /label2/ NP
199         Shared INTEGER /label3/ NUMDONE
200         Shared INTEGER /label4/ Beta,BetaP,N
201         Shared INTEGER /label5/ PIVCOL
202         Shared REAL /label7/ RHS(2000)
203         Shared REAL /label8/ Y(2000)
204         Shared CHARACTER /label11/ NUMLCK
```

```
205       INTEGER ICFret
206       INTEGER MYPIV, MYPIV2
207       INTEGER tempmax
208
209       MYPIV = 1
210       DO 100 I = 1, N
211         tempmax = max(I-Beta,1)
212 C        Compute the amount to subtract from the RHS
213         PriSUM = 0
214         DO 110 MYPIV2 = 1, NUMCOLS
215           J = ASSIGN(MYPIV2)
216           IF ((J.ge.tempmax).and.(J.lt.I)) THEN
217             PriSUM = PriSUM + A((BetaP*MYPIV2)+I-J+1)*Y(J)
218           END IF
219 110       CONTINUE
220 C         Update the RHS
221         CALL CFlock(ICFret,1,'NUMLCK')
222         RHS(I) = RHS(I) - PriSUM
223         NUMDONE = NUMDONE + 1
224         CALL CFulck(ICFret,1,'NUMLCK')
225 C         If I own column I then compute Y(I)
226         IF (I.eq.ASSIGN(MYPIV)) THEN
227           WHEN (NUMDONE.eq.NP) CONTINUE
228           Y(I) = RHS(I) / A((BetaP*MYPIV)+1)
229           MYPIV = MYPIV + 1
230           NUMDONE = 0
231           PIVCOL = PIVCOL + 1
232         ELSE
233           WHEN (PIVCOL.eq.I) CONTINUE
234         END IF
235 100     CONTINUE
236       RETURN
237       END
238
239 C     the bacward solve task using col-sweep
240       SUBROUTINE BACK()
241       INTEGER I,J
242       REAL A(30000)
243       common /pblk1/ A(30000)
244       INTEGER ASSIGN(500)
245       common /pblk2/ ASSIGN(500)
246       INTEGER NUMCOLS
247       common /pblk3/ NUMCOLS
248       Shared INTEGER /label2/ NP
249       Shared INTEGER /label4/ Beta,BetaP,N
250       Shared INTEGER /label5/ PIVCOL
251       Shared REAL /label8/ Y(2000)
252       Shared REAL /label9/ X(2000)
253       Shared CHARACTER /label11/ NUMLCK
254       INTEGER MYPIV, MYPIV2
255       INTEGER tempmax
256
257       MYPIV = NUMCOLS
258       DO 100 J = N, 1, -1
259 C       If this proc owns column J then compute X(J)
```

```
260          IF (J.eq.ASSIGN(MYPIV)) THEN
261            X(J) = Y(J) / A((BetaP*MYPIV)+1)
262            MYPIV = MYPIV - 1
263            IF (MYPIV.eq.0) MYPIV = 1
264            PIVCOL = PIVCOL - 1
265          END IF
266          WHEN (PIVCOL.le.J) CONTINUE
267          tempmax = max(J-Beta,1)
268          DO 110 MYPIV2 = NUMCOLS, 1, -1
269            I = ASSIGN(MYPIV2)
270            IF ((I.le.J-1).and.(I.ge.tempmax)) THEN
271              Y(I) = Y(I) - A((BetaP*MYPIV2)+J-I+1)*X(J)
272            END IF
273 110      CONTINUE
274 100    CONTINUE
275        RETURN
276        END
```

Appendix 4: ConCurrent FORTRAN - shared memory version

```
1          PROGRAM MAIN
2          Shared INTEGER /label1/ PRCNUM(20)
3    C      PRCNUM holds the physical proc number corresponding the
4    C      the logical proc number
5          Shared INTEGER /label2/ NP
6    C      NP is the number of processors
7          Shared INTEGER /label3/ NUMDONE
8          Shared INTEGER /label4/ Beta,BetaP,N
9    C      Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
10         Shared INTEGER /label5/ PIVCOL
11         Shared REAL /label6/ A(30000)
12   C      A contains the matrix
13         Shared REAL /label7/ RHS(2000)
14   C      RHS is the right-hand side vector
15         Shared REAL /label8/ Y(2000)
16   C      The Y vector in the forward solve
17         Shared REAL /label9/ X(2000)
18   C      X is the solution vector
19         Shared CHARACTER /label11/ NUMLCK
20         EXTERNAL ELCOL
21         EXTERNAL FORW
22         EXTERNAL BACK
23         INTEGER PID(20)
24         INTEGER  I
25         INTEGER  ICFret
26         INTEGER  tempmax, tempmin
27
28   C      Allocate a lock
29         CALL CFgetl(ICFret,'NUMLCK')
30         open(unit=2,cpu=1,file='/usr/u1/mtj/concur/choleski/param.dat')
31   C      Read in the number of processors
32         READ(2,525) NP
33         PRINT *, ' Using ',NP,' processors'
34         DO 15 I = 1, NP
```

```
35          PRCNUM(I) = I + 2
36  15    CONTINUE
37
38  C     Decide whether to read in or build the matrix
39        READ(2,525) I
40        IF (I.eq.0) THEN
41          CALL INITMAT(A,RHS,N,Beta,BetaP)
42        ELSE
43          CALL CRMAT(A,RHS,N,Beta,BetaP)
44        END IF
45        WRITE(6,600) N, Beta
46  600   FORMAT(' Order',I4,' matrix with a semi-bandwidth',I4,'.')
47  525   FORMAT(I4)
48
49        PIVCOL = 0
50  C     Start the factorization processes on each processor
51        NUMDONE = NP
52        COBLOCK
53          DO 150 I = 1, NP
54            PROCESS(PID(i),ELCOL(),PRCNUM(I))
55  150     CONTINUE
56        END COBLOCK
57
58
59        PIVCOL = 0
60  C     Start the forward solve processes on each processor
61        NUMDONE = NP
62        COBLOCK
63          DO 160 I = 1, NP
64            PROCESS(PID(i),FORW(),PRCNUM(I))
65  160     CONTINUE
66        END COBLOCK
67
68
69        PIVCOL = N + 1
70  C     Start the back solve processes on each processor
71        NUMDONE = NP
72        COBLOCK
73          DO 170 I = 1, NP
74            PROCESS(PID(i),BACK(),PRCNUM(I))
75  170     CONTINUE
76        END COBLOCK
77
78  C     Print out the solution vector
79        DO 500 J = 1, N
80          WRITE(8,680) J, X(J)
81  500   CONTINUE
82  680   FORMAT(' X(',I4,') = ',6E13.6)
83
84        CALL CFkill(ICFret,0)
85        END
86
87  C     The factorization task
88        SUBROUTINE ELCOL()
89        INTEGER MYNUM
```

```
90        INTEGER K,I,J
91        Shared INTEGER /label2/ NP
92        Shared INTEGER /label3/ NUMDONE
93        Shared INTEGER /label4/ Beta,BetaP,N
94        Shared INTEGER /label5/ PIVCOL
95        Shared REAL /label6/ A(20000)
96        Shared CHARACTER /label11/ NUMLCK
97        INTEGER ICFret
98        INTEGER MYPIV
99        INTEGER plself
100       INTEGER tempmin
101
102 C      Start the choleski factorization loop
103 C      Find out what processor I am
104       MYNUM = plself()
105       MYPIV = MYNUM
106       DO 100 K = 1, N
107         tempmin = min(K+Beta,N)
108 C        If I own the pivot column then compute it
109         IF (K.eq.MYPIV) THEN
110           WHEN (NUMDONE.eq.NP) CONTINUE
111           A((K*BetaP)+1) = sqrt(A((K*BetaP)+1))
112           DO 110 S = K+1, tempmin
113             A((K*BetaP)+S-K+1) = A((K*BetaP)+S-K+1) / A((K*BetaP)+1)
114 110         CONTINUE
115           MYPIV = MYPIV + NP
116           NUMDONE = 0
117           PIVCOL = PIVCOL + 1
118         ELSE
119           WHEN (PIVCOL.eq.K) CONTINUE
120         ENDIF
121 C        Update the rest of the columns
122         DO 120  J = K+MYNUM, tempmin, NP
123           Do 130 I = J, tempmin
124             A((J*BetaP)+I-J+1) = A((J*BetaP)+I-J+1)
125     C                          - A((K*BetaP)+I-K+1)*A((K*BetaP)+J-K+1)
126 130         CONTINUE
127 120       CONTINUE
128         CALL CFlock(ICFret,1,'NUMLCK')
129         NUMDONE = NUMDONE + 1
130         CALL CFulck(ICFret,1,'NUMLCK')
131 100     CONTINUE
132       RETURN
133       END
134
135 C      The forward solve task (using col-sweep)
136       SUBROUTINE FORW()
137       INTEGER MYNUM
138       INTEGER I,J
139       Shared INTEGER /label2/ NP
140       Shared INTEGER /label3/ NUMDONE
141       Shared INTEGER /label4/ Beta,BetaP,N
142       Shared INTEGER /label5/ PIVCOL
143       Shared REAL /label6/ A(20000)
144       Shared REAL /label7/ RHS(2000)
```

```
145       Shared REAL /label8/ Y(2000)
146       Shared CHARACTER /label11/ NUMLCK
147       INTEGER ICFret
148       INTEGER MYPIV
149       INTEGER plself
150       INTEGER tempmin
151
152 C     Find out which processor I am
153       MYNUM = plself()
154       MYPIV = MYNUM
155       DO 100 J = 1, N
156         tempmin = min(J+Beta,N)
157 C       If I am responsible for col J then compute Y(J)
158         IF (J.eq.MYPIV) THEN
159           WHEN (NUMDONE.eq.NP) CONTINUE
160           Y(J) = RHS(J) / A((BetaP*J)+1)
161           MYPIV = MYPIV + NP
162           NUMDONE = 0
163           PIVCOL = PIVCOL + 1
164         ELSE
165           WHEN (PIVCOL.eq.J) CONTINUE
166         ENDIF
167         DO 310 I = J+MYNUM, tempmin, NP
168           RHS(I) = RHS(I) - A((BetaP*J)+I-J+1)*Y(J)
169 310     CONTINUE
170         CALL CFlock(ICFret,1,'NUMLCK')
171         NUMDONE = NUMDONE + 1
172         CALL CFulck(ICFret,1,'NUMLCK')
173 100   CONTINUE
174       RETURN
175       END
176
177 C     The back solve task using col-sweep
178       SUBROUTINE BACK()
179       INTEGER MYNUM
180       INTEGER I,J
181       Shared INTEGER /label2/ NP
182       Shared INTEGER /label3/ NUMDONE
183       Shared INTEGER /label4/ Beta,BetaP,N
184       Shared INTEGER /label5/ PIVCOL
185       Shared REAL /label6/ A(20000)
186       Shared REAL /label8/ Y(2000)
187       Shared REAL /label9/ X(2000)
188       Shared CHARACTER /label11/ NUMLCK
189       INTEGER ICFret
190       INTEGER MYPIV
191       INTEGER plself
192       INTEGER tempmax
193
194 C     find out which processor I am
195       MYNUM = plself()
196       MYPIV = N + 1 - MYNUM
197       DO 100 J = N, 1, -1
198         tempmax = max(J-Beta,1)
199 C       If I am responsible for col J then compute X(J)
```

```
200          IF (J.eq.MYPIV) THEN
201            WHEN (NUMDONE.eq.NP) CONTINUE
202            X(J) = Y(J) / A((BetaP*J)+1)
203            MYPIV = MYPIV - NP
204            NUMDONE = 0
205            PIVCOL = PIVCOL - 1
206          ELSE
207            WHEN (PIVCOL.eq.J) CONTINUE
208          ENDIF
209          DO 410 I = J-MYNUM, tempmax, -NP
210            Y(I) = Y(I) - A((BetaP*I)+J-I+1)*X(J)
211 410      CONTINUE
212          CALL CFlock(ICFret,1,'NUMLCK')
213          NUMDONE = NUMDONE + 1
214          CALL CFulck(ICFret,1,'NUMLCK')
215 100    CONTINUE
216        RETURN
217        END
```

Appendix 5: PISCES message passing version

```
1          tasktype chol
2          integer N, M, P, Beta, BetaP
3  C        N is the matrix size, M is the max number of columns per proc
4  C        P is the max number or processors, Beta is the semi-bandwidth,
5  C        BetaP is Beta + 1
6          integer MaxN, MaxBetaP
7  C        MaxN is the max matrix size, MaxBetaP is the max semi-bandwidth
8          parameter (MaxN=305)
9          parameter (MaxBetaP=80)
10         parameter (P=25)
11 * M should be at least as great as N/P
12         parameter (M=305)
13         integer colasgn(P,M)
14 C        colasgn is the array of which columns a processor owns
15         integer numcols(P)
16 C        numcols is the number of columns owned by each processor
17         taskid tasknum(P)
18 C        the array of task id's
19         common /tblk/tasknum(P)
20         handler getid
21         real X(MaxN)
22         handler mnewx
23         integer xok
24         common /bblk1/xok
25         real Curx
26         common /bblk2/Curx
27         real A(MaxN,MaxBetaP)
28 C        A is the matrix
29         common /result/A(MaxN,MaxBetaP)
30         real B(MaxN)
31 C        B is the right hand side
32         common /rhs/B(MaxN)
33         integer owner(MaxN)
```

```
34 C       the array of who owns each column
35         signal fordon
36         integer numclust
37         integer clust
38         handler newcol
39         enddeclarations
40 *
41 *  Generate test matrices
42 *
43         CALL SETCPU(1)
44         open(unit=2,file='/usr/u1/mtj/pisces/mchol3/param.dat')
45         READ(2,500) N
46         print *, ' N = ',N
47         READ(2,500) Beta
48         print *, ' Beta = ',Beta
49 500     FORMAT(I4)
50         BetaP = Beta + 1
51         do 10 i = 1,N
52           A(i,1) = Beta * 4.0
53           do 20 j = 2,Beta+1
54             A(i,j) = -1.0
55 20        continue
56 10      continue
57 *
58 * Make the assignment of columns to tasks
59 *
60         clust=pppcmin()
61         numclust = 0
62         do 50 i = 1, 100000
63           numcols(clust) = 0
64           clust = pppcnxt(clust)
65           numclust = numclust + 1
66           if (clust.eq.pppcmin()) goto 55
67 50      continue
68 55      continue
69         myclust = pppgclu (pppself)
70         clust = pppcmin()
71         do 60 i = 1,N
72 *          Skip the cluster on which this task is running
73           if (myclust .eq. clust) clust = pppcnxt (clust)
74           numcols(clust) = numcols(clust) + 1
75           colasgn(clust,numcols(clust)) = i
76           owner(i) = clust
77           clust = pppcnxt(clust)
78  60      continue
79 *
80 * Make the assignment of tasks to clusters
81 *
82         clust=pppcmin()
83         do 70 i = 1, 100000
84           if (myclust .eq. clust) clust = pppcnxt (clust)
85           on cluster(clust) initiate colsrv (N,Beta,numclust,
86 &                    pppv1 (numcols, clust, clust),
87 &                    pppm1 (colasgn, P, M, clust, clust, 1, M),
88 &                    pppv1 (owner,1,N))
```

```
89            clust = pppcnxt(clust)
90            if (clust.eq.pppcmin()) goto 75
91  70      continue
92  75      continue
93  *
94  * Get the taskid of every task
95  *
96          accept numclust-1 of
97            getid
98          endaccept
99  *       Send the collection of taskid's to every task
100         to all send allids(pppv1(tasknum,1,P))
101 *
102 * Send the columns that are assigned to each processor to that processor
103 *
104         clust=pppcmin()
105         do 80 i = 1, 100000
106           if (myclust .eq. clust) clust = pppcnxt (clust)
107           do 90 j = 1, numcols(clust)
108             to tasknum(clust) send incol
109 &               (j,BetaP,pppm1(A,MaxN,MaxBetaP,colasgn(clust,j),
110 &               colasgn(clust,j),1,BetaP))
111 90          continue
112           clust = pppcnxt(clust)
113           if (clust.eq.pppcmin()) goto 85
114 80      continue
115 85      continue
116 *
117 * Wait for results to come back
118 *
119         accept N of
120           newcol
121         endaccept
122 *
123 * Initialize the RHS to all 1's
124 *
125         do 120 i=1,N
126           B(i) = 1.0
127 120     continue
128 *
129 * Start the forward solve
130 *
131         do 130 i=1,N
132           to tasknum(owner(i)) send bval(B(i))
133 130     continue
134         accept numclust-1 of
135           fordon
136         endaccept
137 *
138 * Start back solve
139 *
140         accept n of
141           mnewx
142         endaccept
143 *
```

```
144 * print the solution vector
145 *
146       do 450 i=1,N
147       WRITE (8,650) i,X(i)
148 450    continue
149 650    FORMAT(' X(',I4,') = ',E13.6)
150       terminate
151       end
152 *
153 * HANDLER: Store the taskid in the array
154 *
155       handler getid (index, tasknum(index))
156       integer index
157       integer P
158       parameter (P=25)
159       taskid tasknum(P)
160       common /tblk/tasknum(P)
161       enddeclarations
162       return
163       end
164 *
165 * HANDLER: Store the incoming column in the array
166 *
167       handler newcol (col, BetaP,
168 &          pppm1(A,MaxN,MaxBetaP,col,col,1,BetaP))
169       integer MaxN, MaxBetaP, BetaP
170       parameter (MaxN=305)
171       parameter (MaxBetaP=80)
172       real A(MaxN,MaxBetaP)
173       common /result/A(MaxN,MaxBetaP)
174       integer col
175       enddeclarations
176       return
177       end
178 *
179 * factorization, back solve and forward solve task
180 *
181       tasktype colsrv (N, Beta, numclust, numcols,
182 &                pppv1 (mycols, 1, M), pppv1(owner, 1, N))
183 *     These parameter must match that in the chol tasktype definition
184       integer M, BetaP
185       integer MaxBetaP
186       parameter (M=305)
187       parameter (MaxBetaP=80)
188       integer P
189       parameter (P=25)
190       integer MaxN
191       parameter (MaxN=305)
192       integer owner(MaxN)
193       integer numclust
194       integer N, Beta, numcols
195       common /mblk1/ numcols
196       integer mycols(M)
197       common /mblk2/ mycols(M)
198       handler incol
```

```
199       handler pivot
200       real Amine(M,MaxBetaP)
201 C      Amine contains the columns that this processor owns
202       common /blk1/Amine(M,MaxBetaP)
203       real piv(MaxBetaP)
204       common /blk2/piv(MaxBetaP)
205       integer pivnum
206       common /blk3/pivnum
207       integer curk
208       integer curin
209       common /blk5/curin
210       real Ymine(M)
211 C      Ymine contains the Y values that this processor owns
212       common /blk4/Ymine(M)
213       real sum
214       integer sent
215       integer k,s,i
216       handler bval
217       handler bup
218       handler newx
219       integer xok
220       common /bblk1/xok
221       real Curx
222       common /bblk2/Curx
223       real B(M)
224 C      B contains the right hand side values this processor owns
225       common /fblk1/ B(M)
226       integer bcount(M)
227 C      bcount contains the number of updates to B(i) received
228       common /fblk2/ bcount(M)
229       taskid tasknum(P)
230       common /tblk/tasknum(P)
231       handler allids
232       enddeclarations
233
234       BetaP = Beta + 1
235       myclust = pppgclu (pppself)
236 *
237 *      Send my taskid to the parent
238 *
239       to parent send getid(myclust,pppself)
240 *      Accept the vector of taskids
241       accept 1 of
242         allids
243       endaccept
244 *      receive the columns that we are assigned
245       accept numcols of
246         incol
247       endaccept
248 *
249 *      Begin the factorization
250 *
251       myk=1
252       do 10 k=1,N
253 *        if I own column k then compute and broadcast the pivot
```

```
254           if (mycols(myk).eq.k) THEN
255             Amine(myk,1)=sqrt(Amine(myk,1))
256             do 20 s=2,(min(k+Beta,N)-k+1)
257               Amine(myk,s)=Amine(myk,s)/Amine(myk,1)
258 20          continue
259             to all send pivot(mycols(myk), BetaP,
260 &               pppm1(Amine,M,MaxBetaP,myk,myk,1,BetaP))
261             to parent send newcol(mycols(myk), BetaP,
262 &               pppm1(Amine,M,MaxBetaP,myk,myk,1,BetaP))
263             do 30 s=1,BetaP
264               piv(s)=Amine(myk,s)
265 30          continue
266             myk = myk + 1
267           ELSE
268 40          accept 1 of
269               pivot
270             endaccept
271 *           if a pivot column is received out of order then
272 *           send it back to myself and get another
273             if (pivnum.ne.k) THEN
274               to self send pivot(pivnum, BetaP,
275 &                 pppv1(piv,1,BetaP))
276               goto 40
277             ENDIF
278           ENDIF
279 *         update the rest of the columns that I own
280           do 50 s = myk,numcols
281             if ((mycols(s).gt.k).and.(mycols(s).le.min(k+Beta,N)))
282 &             THEN
283               do 60 i=1,min(Beta+k,N)-mycols(s)+1
284                 Amine(s,i)=Amine(s,i)-piv(i+mycols(s)-k)*
285 &                   piv(mycols(s)-k+1)
286 60              continue
287             ENDIF
288 50        continue
289 10      continue
290 *
291 * start forward solve (using inner product)
292 *
293         curin = 1
294 *       receive the right hand side values that I own
295         accept numcols of
296           bval
297         endaccept
298         do 90 i = 1, numcols
299           bcount(i) = 0
300 90      continue
301         curin = 1
302         do 100 i = 1, N
303           sum = 0
304           sent = 0
305           do 110 s = 1, numcols
306             if ((mycols(s).ge.max(1,i-Beta)).and.(mycols(s).lt.i))
307 &             THEN
308               sum = sum + Amine(s,i-mycols(s)+1) * Ymine(s)
```

```
309             ELSE if (mycols(s).eq.i) THEN
310 *             if I own this column then wait for everyone
311 *             to send me the updates and then compute Y(I)
312               b(s) = b(s) - sum
313               sent = 1
314 150           if (bcount(s).eq.numclust-2) goto 160
315               accept all of
316                 bup
317               endaccept
318               goto 150
319 160           continue
320               Ymine(curin) = B(curin) / Amine(curin,1)
321               curin = curin + 1
322             ENDIF
323 110       continue
324           if (sent.eq.0) then
325             to tasknum(owner(i)) send bup(i,sum)
326           endif
327 100     continue
328         to parent send fordon
329 *
330 * start backsolve (using col-sweep)
331 *
332         curk=numcols
333         do 200 j=N,1,-1
334           if (curk.eq.0) goto 300
335           if (mycols(curk).eq.j) THEN
336             Curx=Ymine(curk)/Amine(curk,1)
337             to all send newx(j,Curx)
338             to parent send mnewx(j,Curx)
339             curk = curk - 1
340           ELSE
341             xok = j
342 210         accept 1 of
343               newx
344             endaccept
345             if (xok.ne.0) goto 210
346           ENDIF
347           do 220 s = curk, 1, -1
348             if ((mycols(s).lt.j).and.(mycols(s).ge.max(1,j-Beta)))
349 &             THEN
350               Ymine(s) = Ymine(s) - Amine(s,j-mycols(s)+1)*Curx
351             ENDIF
352 220       continue
353 200     continue
354 300     continue
355         terminate
356         end
357 *
358 *       HANDLER: receive a column and place it in Amine
359 *
360         handler incol(col, BetaP,
361 &           pppm1(Amine,M,MaxBetaP,col,col,1,BetaP))
362         integer M, BetaP
363         parameter (M=305)
```

```
364         integer MaxBetaP
365         parameter (MaxBetaP=80)
366         real Amine(M,MaxBetaP)
367         common /blk1/Amine(M,MaxBetaP)
368         integer col
369         enddeclarations
370         return
371         end
372 *
373 *       HANDLER: receive a pivot column and place it in piv
374 *
375         handler pivot(pivnum, BetaP, pppv1(piv,1,BetaP))
376         integer BetaP
377         integer MaxBetaP
378         parameter (MaxBetaP=80)
379         real piv(MaxBetaP)
380         common /blk2/piv(MaxBetaP)
381         integer pivnum
382         common /blk3/pivnum
383         enddeclarations
384         return
385         end
386 *
387 * HANDLER: take in the updated bval
388 *
389         handler bval (B(curin))
390         integer M
391         parameter (M=305)
392         real B(M)
393         common /fblk1/ B(M)
394         integer curin
395         common /blk5/curin
396         enddeclarations
397         curin = curin + 1
398         return
399         end
400 *
401 * HANDLER: take in the new x and place in x(row) for main task
402 *
403         handler mnewx (row,X(row))
404         integer row
405         parameter (MaxN=460)
406         real X(MaxN)
407         common /xblk/X(MaxN)
408         enddeclarations
409         return
410         end
411 *
412 * HANDLER: take in the new x
413 *
414         handler newx (row,Curx)
415         integer row
416         integer xok
417         common /bblk1/xok
418         real Curx
```

```
419         common /bblk2/Curx
420         enddeclarations
421
422         if (row.eq.xok) THEN
423           xok = 0
424         ELSE
425           to self send newx(row,Curx)
426         ENDIF
427         return
428         end
429 *
430 * HANDLER: Update the rhs
431 *
432         handler bup (row, bval)
433         integer MaxN,M
434         parameter (M=305)
435         parameter (MaxN=305)
436         integer row
437         real bval
438         real B(M)
439         common /fblk1/ B(M)
440         integer bcount(M)
441         common /fblk2/ bcount(M)
442         integer numcols
443         common /mblk1/ numcols
444         integer mycols(M)
445         common /mblk2/ mycols(M)
446         enddeclarations
447
448         do 10 i=1, numcols
449           if (row.eq.mycols(i)) goto 15
450 10       continue
451 15       continue
452         B(i) = B(i) - bval
453         bcount(i) = bcount(i) + 1
454         return
455         end
456 *
457 * HANDLER: accepts vector of taskids
458 *
459         handler allids (pppv1(tasknum,1,P))
460         integer P
461         parameter (P=25)
462         taskid tasknum(P)
463         common /tblk/tasknum(P)
464         enddeclarations
465         return
466         end
```

Appendix 6: PISCES FORCE version

```
1         tasktype chol
2         INTEGER  I,J,K,S
3         INTEGER tempmin
```

```
4         INTEGER tempmax
5         shared
6           INTEGER Beta,BetaP,N
7   C       Beta is the semi-bandwidth, BetaP is Beta+1, N is the matrix size
8           REAL A(20000)
9   C       A contains the matrix
10          REAL RHS(1000)
11  C       RHS is the right-hand side vector
12          REAL Y(1000)
13  C       The Y vector in the forward solve
14          REAL X(1000)
15  C       X is the solution vector
16          INTEGER STARTTIME,ENDTIME
17          common /blk1/Beta,BetaP,N,A
18          common /blk2/RHS,Y,X,STARTTIME,ENDTIME
19        end shared
20        end declarations
21
22        forcesplit
23        barrier
24  C       Decide whether to read in or build the matrix
25          CALL SETCPU(1)
26          open(unit=9,file='/usr/u1/mtj/pisces/fchol/param.dat')
27          READ(9,525) I
28          IF (I.eq.0) THEN
29            CALL INITMAT(A,RHS,N,Beta,BetaP)
30          ELSE
31            CALL CRMAT(A,RHS,N,Beta,BetaP)
32          END IF
33          WRITE(6,600) N, Beta
34  600     FORMAT(' Order',I4,' matrix with a semi-bandwidth',I4,'.')
35  525     FORMAT(I4)
36        end barrier
37
38  C     Start the choleski factorization loop
39        DO 100 K = 1, N
40  C       Compute the first element of the pivot column
41          barrier
42            A((K*BetaP)+1) = sqrt(A((K*BetaP)+1))
43          end barrier
44          tempmin = min(K+Beta,N)
45  C       Compute the rest of the pivot column
46          presched do 110 S = K+1, tempmin
47            A((K*BetaP)+S-K+1) = A((K*BetaP)+S-K+1) / A((K*BetaP)+1)
48  110     CONTINUE
49          barrier
50          end barrier
51  C       Update the rest of the columns
52          presched do 120  J = K+1, tempmin
53            Do 130 I = J, tempmin
54              A((J*BetaP)+I-J+1) = A((J*BetaP)+I-J+1)
55      C                          - A((K*BetaP)+I-K+1)*A((K*BetaP)+J-K+1)
56  130       CONTINUE
57  120     CONTINUE
58  100   CONTINUE
```

```
59
60 C     The foward solve (using col-sweep)
61       DO 300 J = 1, N
62         barrier
63         Y(J) = RHS(J) / A((BetaP*J)+1)
64         end barrier
65         tempmin = min(J+Beta,N)
66         presched do 310 I = J+1, tempmin
67           RHS(I) = RHS(I) - A((BetaP*J)+I-J+1)*Y(J)
68 310     CONTINUE
69 300    CONTINUE
70
71 C     The backward solve (using col-sweep)
72       DO 400 J = N, 1, -1
73         barrier
74         X(J) = Y(J) / A((BetaP*J)+1)
75         end barrier
76         tempmax = max(J-Beta,1)
77         presched do 410 I = J-1, tempmax, -1
78           Y(I) = Y(I) - A((BetaP*I)+J-I+1)*X(J)
79 410     CONTINUE
80 400    CONTINUE
81
82 C     Print out the solution vector
83       barrier
84       DO 500 J = 1, N
85         WRITE(8,680) J, X(J)
86 500    CONTINUE
87 680    FORMAT(' X(',I4,') = ',6E13.6)
88       end barrier
89
90       terminate
91       end
```

Appendix 7

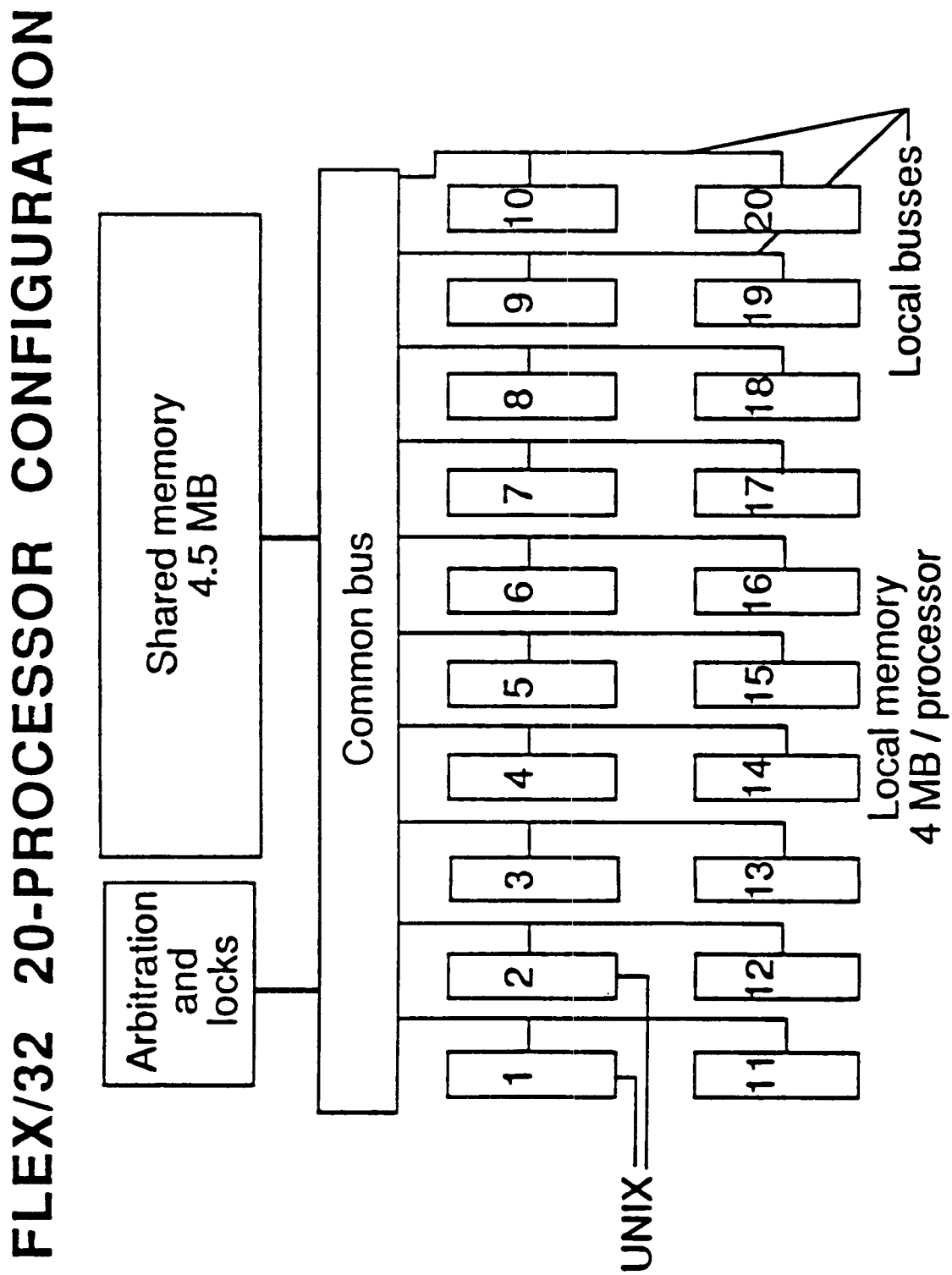
Table of execution times in seconds
for Factorization, Forward Solve and Back Solve
(Matrix dimension = 456, Bandwidth = 112)

Language	PE's	Fact	Forw	Back
Force (Local)				
"	1	201.80	6.00	7.38
"	2	105.00	3.06	3.76
"	4	56.62	1.66	2.00
"	8	33.38	1.06	1.22
"	16	21.04	0.94	1.04
Force (Shared)				
"	1	239.94	3.04	3.14
"	2	121.82	1.62	1.68
"	4	62.22	0.94	0.98
"	8	32.56	0.68	0.72
"	16	18.46	0.92	0.94
Concur (Local)				
"	1	180.52	8.46	7.28
"	2	96.22	7.44	5.86
"	4	55.66	10.36	7.56
"	8	41.44	17.62	14.14
"	16	44.28	30.88	28.08
Concur (Shared)				
"	1	217.70	5.76	5.98
"	2	116.76	6.04	6.10
"	4	68.04	7.38	7.50
"	8	49.26	17.48	17.50
"	16	51.20	30.96	30.98
Pisces (Message-passing)				
"	1	287.22	14.38	9.44
"	2	161.04	12.74	7.34
"	4	97.78	7.80	5.10
"	8	67.30	5.36	4.70
"	16	55.14	4.18	5.20
Pisces (Force)				
"	1	225.50	3.04	3.08
"	2	114.24	1.68	1.70
"	4	58.72	1.12	1.16
"	8	30.80	0.86	0.86
"	16	17.56	0.98	0.98

Appendix 8

Chart of lines of code (all i/o and comments removed)			
Lines	Words	Chars	Program
92	305	2846	Force (Local)
48	170	1525	Force (Shared)
214	649	6460	ConCurrent (Local)
161	501	4784	ConCurrent (Shared)
344	859	9706	Pisces (Message-passing)
53	157	1572	Pisces (Force)

Appendix 9


FLEX/32 20-PROCESSOR CONFIGURATION
Shared memory
4.5 MB
Arbitration
and
locks
Common bus
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
UNIX
Local memory
4 MB / processor
Local busses

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181779 ICASE Report No. 89-6		

4. Title and Subtitle	5. Report Date
A LANGUAGE COMPARISON FOR SCIENTIFIC COMPUTING ON MIMD ARCHITECTURES	January 1989
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Mark T. Jones Merrell L. Patrick Robert G. Voigt	89-6
	10. Work Unit No. 505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18107 NAS1-18605
	13. Type of Report and Period Covered

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

16. Abstract

Choleski's method for solving banded symmetric, positive definite systems is implemented on a multiprocessor computer using three FORTRAN based parallel programming languages, the Force, PISCES and Concurrent FORTRAN. The capabilities of the language for expressing parallelism and their user friendliness are discussed, including readability of the code, debugging assistance offered, and expressiveness of the languages. The performance of the different implementations is compared. It is argued that PISCES, using the Force for medium-grained parallelism, is the approprate choice for programming Choleski's method on the multiprocessor computer, Flex/32.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
parabolic equations, linear quadratic regulator problem, stabilizability	Unclassified - Unlimited 61 - Computer Programming & Software

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	40	A03

NASA FORM 1626 OCT 86

NASA-Langley, 1989